TLB For Free:

In-Cache Address Translation For A Multiprocessor Workstation

Scott Allen Ritchie

Computer Science Division
Electrical Engineering and Computer Sciences Department
University of California, Berkeley
Berkeley, CA 94720

Abstract: In the design of SPUR, a high-performance multiprocessor workstation, the need for large "snooping" caches suggests a new approach to virtual address translation. By performing this translation in each processor's virtual cache, the need for separate translation lookaside buffers is eliminated. Trace-driven simulations show that normal cache behavior is only minimally effected, and that unless an extremely large and complex TLB were built, using a separate device would actually reduce system performance.

Key Words And Phrases: Address Space, Address Translation, Snooping Cache, Translation Lookaside Buffer (TLB), Multiprocessor Workstation, Virtual Memory, VLSI.

May 13, 1985

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **13 MAY 1985**	2. REPORT TYPE	3. DATES COVERED **00-00-1985 to 00-00-1985**

4. TITLE AND SUBTITLE **TLB For Free: In-Cache Address Translation For A Multiprocessor Workstation**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of California at Berkeley,Department of Electrical Engineering and Computer Sciences,Berkeley,CA,94720**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR’S ACRONYM(S)
	11. SPONSOR/MONITOR’S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
In the design of SPUR, a high-performance multiprocessor workstation, the need for large "snooping" caches suggests a new approach to virtual address translation. By performing this translation in each processor’s virtual cache, the need for separate translation lookaside buffers is eliminated. Trace-driven simulations show that normal cache behavior is only minimally effected, and that unless an extremely large and complex TLB were built, using a separate device would actually reduce system performance.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**38**	

TLB For Free:

In-Cache Address Translation For A Multiprocessor Workstation†

Scott Allen Ritchie

Computer Science Division
Electrical Engineering and Computer Sciences Department
University of California, Berkeley
Berkeley, CA 94720

1. Introduction

As early as the 1960's, computer systems were providing users with the abstraction known as *virtual memory*. First appearing in the Atlas computer [Foth61], virtual memory eliminated the need for program overlays by automatically transferring data to and from "backing store". Programmers were given the illusion of a much larger address space and programs were now independent of the size of memory. The authors of MULTICS called virtual memory "generalized addressing" [Dale68] and demonstrated that the use of separate address spaces could provide protection and sharing in a controlled fashion.

Machines lacking virtual memory are considered to be severely limited. The migration from physical to virtual memory has taken place in computers at all levels. Of mainframes we saw the progression from IBM S/360 to S/370 [Case78], in minicomputers from the DEC PDP-11 to VAX-11 [Stre78], and among workstations from the Xerox Alto to the Dorado [Pier83]. In each case, virtual memory was cited as one chief feature of the new architecture. As computer architects consider the structure of future systems, they must face the issue of how to support virtual memory.

Using multiple processors to obtain higher performance from single-user workstations is one area of recent research in computer architecture. This is one topic being explored by the SPUR (Symbolic Processing Using RISCs) project at Berkeley. Machines of this type are likely to be characterized by faster processors and more physical memory than was previously feasible. Large multiple virtual address spaces will be provided for protection and sharing among programs. Lastly, the multiprocessor nature of these systems dictates the need for the consistency of shared data, including address translation information.

† The material presented here is based on research sponsored by the Defense Advanced Research Projects Agency under the SPUR (Symbolic Processing Using RISCs) project.

This paper examines some of the existing means of hardware support for the translation of virtual to physical addresses. These methods are then evaluated in light of the SPUR multiprocessor RISC project. The proposal for SPUR uses the existing caches as translation buffers, doing away with the need to build a separate device. It also solves the problem of data consistency for translation information. Trace-driven simulations show that the cost of a separate translation buffer cannot be justified. These results suggest that unless a large and complex TLB were built, this separate device would actually *reduce* performance.

Although the motivation for this study arises from multiprocessors, the results are equally valid for uniprocessors. In addition, translation mechanisms are typically understood and evaluated for a single processor. Therefore a uniprocessor model will be assumed unless the effects of multiprocessing are relevant.

2. Brief Survey of Existing Translation Mechanisms

To provide for virtual addressing, memory is divided into fixed-size blocks called *pages* that can then be relocated both in primary and disk storage. Virtual addresses used by programs must be translated, or *mapped*, into physical addresses before memory may be accessed. Figure 2.1 shows that the high-order bits of the virtual address must be converted by some means, from the *virtual page number*, into the corresponding *physical page number*. Note that the offset

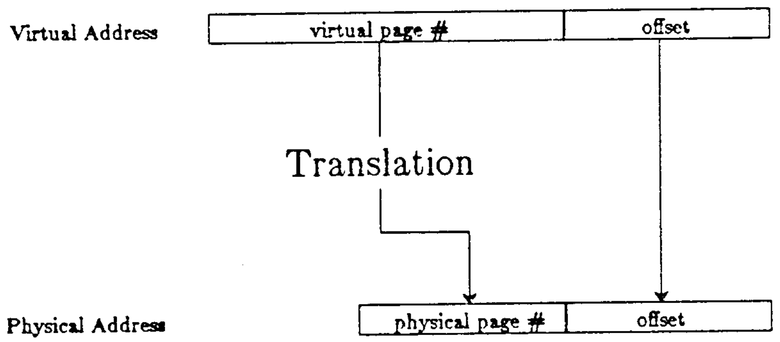


Figure 2.1: The General Translation Process

By some combination of hardware and software lookup, the translation process converts the virtual page number into an address in physical memory. The offset of a particular byte within a page remains unaltered.

bits describing the particular byte within the page are not altered.

Translation normally consists of some form of table lookup using the virtual page number to index into a *page table* maintained by the operating system. An individual *page table entry* (PTE) contains the physical location of the page plus any associated status or protection bits. This lookup could be done entirely in software, but to improve performance, some form of hardware support is usually provided. The subsections below describe some typical methods.

2.1. The Original Translation Scheme

Atlas, the first virtual memory computer, employed a simple translation scheme. A register was associated with each of the 32 pages of core memory. Each register contained the virtual page number of the page stored in that physical page frame. When a reference to memory was made, the virtual page number was compared with each of these page registers in parallel. If there was a match, the word from the appropriate page was sent back to the processor; otherwise, the supervisor was invoked to bring the required page in from drum storage. Given the size of typical memories today, this fully-associative lookup would be prohibitively expensive. A modern memory may contain 256 megabytes of storage, and even with a 4 kilobyte page size the hardware would have to compare over 64,000 addresses in parallel.

2.2. A Fully Resident Memory Map

Perhaps the most intuitive means of hardware support is to keep a page table that maps all virtual pages to the corresponding physical page. Figure 2.2 shows how the page table for a single address space may be kept fully resident in a specially-dedicated "Mapping RAM". The virtual page number addresses the entry in the RAM that contains the physical page number and flag bits for the page.

This approach is used by the Xerox Dorado [Clar81] and is feasible because only one (256Mb) virtual address space is supported. However, with multiple large address spaces, the amount of high-speed RAM required becomes impractical. Since the Dorado virtual page number is 18 bits, the map must have entries for the 256K 1Kb pages. Physical page numbers are 14 bits, so with the flags this requires 17 256K RAM chips: a little over 128K bytes. If the virtual address space were 32 bits, even with a larger 4K byte page, over a million mapping entries would be needed. With more physical memory, the amount of the mapping RAM required can easily exceed the size of the Dorado's original 8 megabyte main memory.

This approach is simple but wasteful. Address spaces are typically used *sparsely*: often only the first and last kilobyte. Furthermore, it is well known that programs tend to exhibit *locality* of reference [Denn72]. There is no need to keep

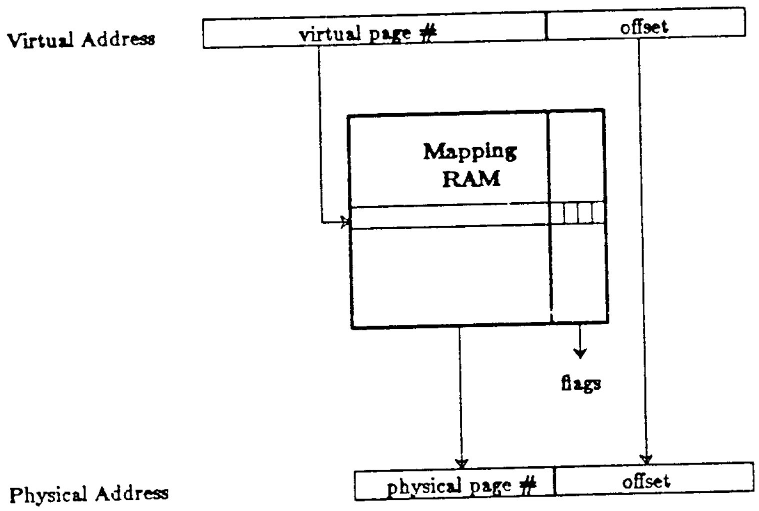


Figure 2.2 : A Fully Resident Mapping

With a single, small address space, it is feasible to keep all of the page mapping in special memory. One RAM access is then sufficient to perform translation.

the entire mapping resident.

Another limiting factor with this design lies in supporting multiple address spaces. With only one space, entries in the map are changed infrequently. With several different address spaces, however, either copies of the mapping hardware must be provided for each space, or all the entries must be *invalidated* when context switches occur. Context switches typically happen on every interrupt, and these occur about 10 to 100 times a second. Invalidating and rewriting roughly one megabyte of RAM this frequently would create severe performance problems.

2.3. Two-level Implementation of a Sparse Mapping

In a machine with multiple address spaces, only one mapping is valid at a time. These multiple address spaces may be considered to be subspaces of one larger *global address space*. Figure 2.3 shows how this is done by extending the virtual address with a a *context* register to identify the current subspace. Since switching occurs between a few active contexts and only a portion of the address space for each is used, a *sparse* mapping of this global address space will suffice. This is achieved by splitting the one RAM in the previous method into separate segment and page maps. The SUN Workstation employs this type of a mechanism [Bech82] and allows for at most 8 loaded contexts.

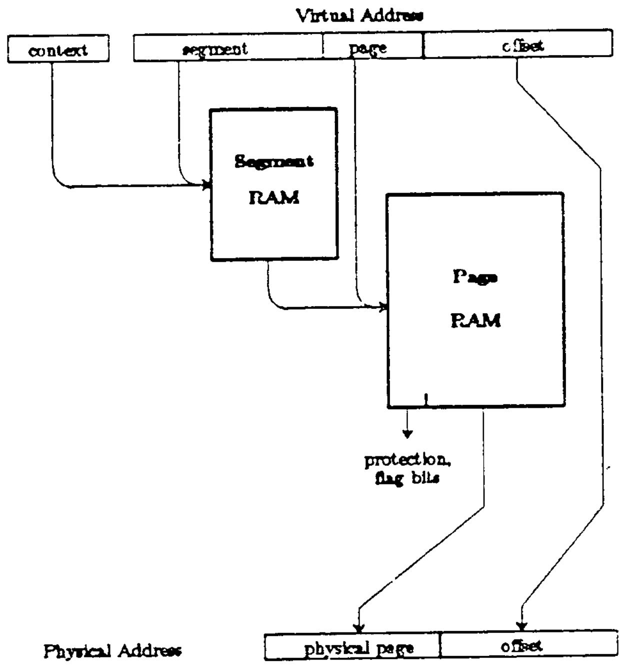


Figure 2.3 : Splitting Mapping RAM Into Two Levels

A 'context' register allows for the distinction between several virtual address spaces. Two RAM accesses provide a sparse mapping of what is effectively one larger global address space. Software must determine what is most appropriate to load into RAM.

This method requires far less RAM than if the entire space were kept resident. For example, the SUN 2.0 uses 2K bytes in the Segment RAM and 8K bytes in the Page RAM to translate the more frequently used portions of multiple address spaces that would require 256K bytes in the previous method. The operating system loads as many of the page table entries for the currently active contexts (processes) as possible. Although many different contexts exist at any point in time, in practice only a few are highly active. When switching between these few active contexts, no invalidation of the map is necessary. Only when a new context becomes active, or an old context gets remapped, must entries be rewritten.

There are several drawbacks to this technique. This now requires two RAM accesses as compared to one in the previous method. Since the "page" field from the virtual address is appended to the segment number to index into Page RAM, pages cannot be mapped individually. SUN uses a four bit field that forces mapping to be in contiguous blocks of 16 pages. Finally, the operating system must successfully predict what portions of address spaces will be the most active to get the best performance.

2.4. Demand Caching of Page Table Entries : A TLB

Perhaps the most familiar of hardware mechanisms is the Translation Lookaside Buffer (TLB)†. The DEC VAX family of computers [DEC 81] and IBM mainframes [SMIT82] use this mechanism. A TLB is a cache memory that automatically stores the page table entries most recently used in translation. This makes the best use of a small amount of memory without software needing to decide *a priori* what will be most useful.

Figure 2.4 shows that the low-order bits of the virtual page number are used to index into the cache. The remaining high-order bits are used as a tag to compare against the cache contents. The figure also shows the most-significant

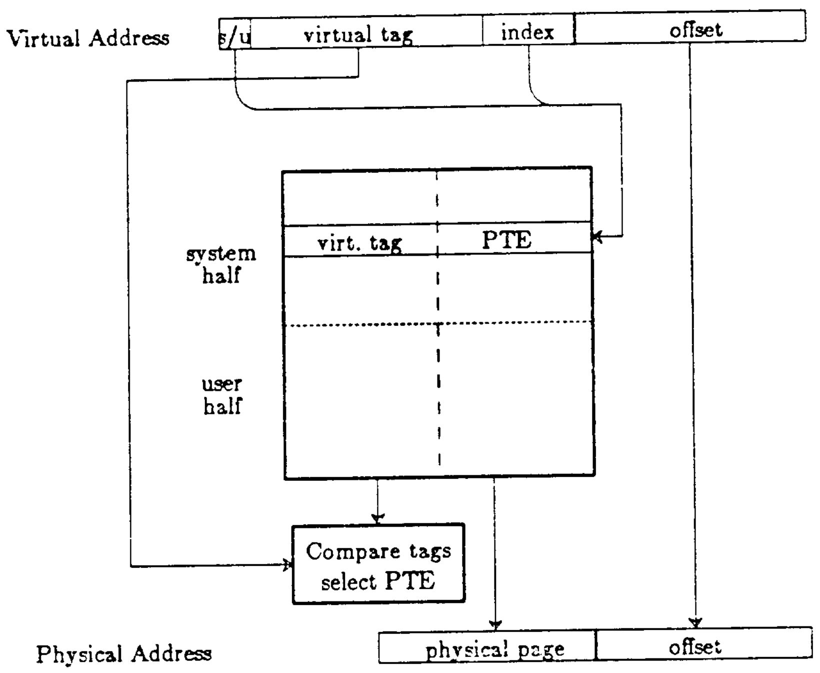


Figure 2.4 : Translation Lookaside Buffer

In a translation lookaside buffer, page table entries are cached as they are used. A division between process and system entries allows only the user-specific portion of the buffer to be flushed on context switch. The cost of this method includes not only the cache access time, but also the comparison of tags.

†This device is also known as a Directory LookAside Table (DLAT) or simply a Translation Buffer.

bit of the virtual address being appended to index into the translation buffer. If this bit distinguishes between separate system and user regions of the address space, then the buffer is effectively divided into these two regions as well. The advantage to this is that on context switch, only the user-specific half of the buffer needs to be invalidated. The switch to a new user's virtual address space requires that the old mapping be purged, while the shared system space remains unchanged. The VAX-11/780 does precisely this [DEC 81].

The ELXSI 6400 has extended the notion of separate user and system divisions of the TLB by providing sixteen copies of the hardware. At any given time, one copy is designated to provide the current user mapping. The switch between user contexts can thus be performed with little invalidation by selecting the appropriate user copy. This is similar to the use of the context register in the SUN two-level mapping scheme. However, ELXSI requires that only one context is used for the system and the remaining 15 are for user processes, while Sun allows the mix of any 8 processes.

More complicated methods of indexing into the translation buffer can yield higher hit rates. Often, selected bits from the virtual address are hashed to form the index. The IBM and Amdahl TLBs both use hashing based on an Exclusive-OR of address bits [Smit82].

Typical translation buffers have 128 to 512 page table entries. For simplicity, the buffer shown in Figure 2.4 has only one entry per row selected by the index: it

Machine	TLB Size (entries)	Number of Sets	Set Size (entries)	Page Size (bytes)	Memory Mapped (bytes)
VAX-11/730	128	128	1	512	64K
VAX-11/780	128	64	2	512	64K
VAX-11/750	512	256	2	512	256K
VAX 8600	512	512	1	512	256K
IBM 370 3033	128	64	2	4K	512K
Amdahl 470V/6	256	128	2	4K	1024K
Amdahl 470V/8	512	256	2	4K	2048K

Table 2.1 : Commercial Translation Lookaside Buffer (TLB) Parameters

The VAX TLBs are separated into two halves: one half is for process space, the other is for system space. The IBM and Amdahl TLBs use a hash function to index into the buffer. For the same number of entries the VAX TLBs map much less memory than IBM and Amdahl because of the smaller VAX page size (See Figure 4.1).

is said to be *direct-mapped*. More often, theses caches have two entries per row and are therefore called *two-way set-associative*. The index selects one row, and comparisons must be performed on both tags. Table 2.1 shows the parameters for several commercial translation buffers.

3. Design of an In-Cache Translation Mechanism

3.1. Overview of a Multiprocessor Workstation

The SPUR multiprocessor, shown in Figure 3.1, will consist of about ten processor-cache pairs on a common bus to shared memory. Two custom VLSI chips form the basis of each processor-cache pair. The first is a 32-bit Reduced

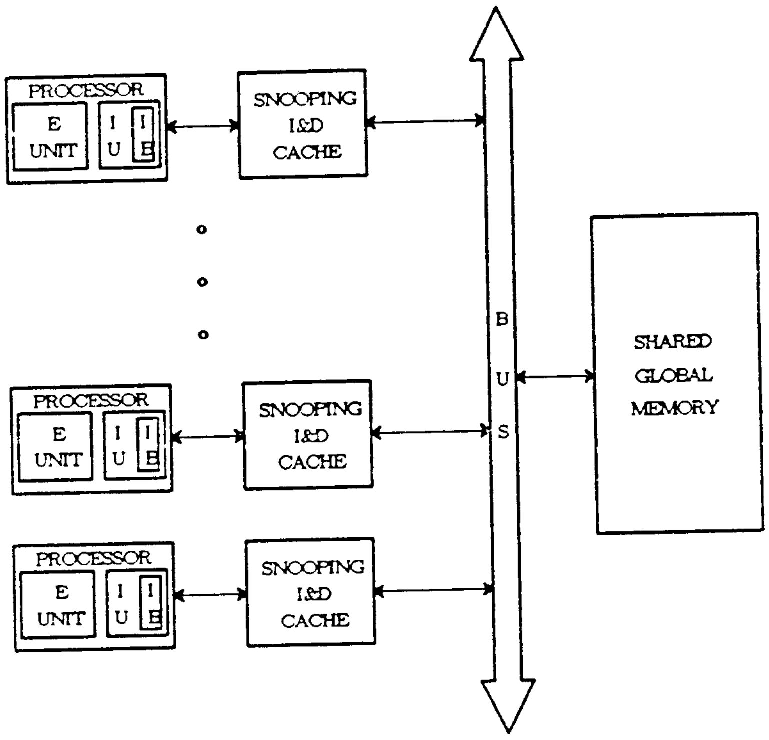


Figure 3.1 : Block Diagram of the SPUR Multiprocessor
Each processor is a RISC with an on-chip instruction buffer (IB). Large virtual caches provide high speed access to instructions and data, greatly reduce bus traffic, and maintain consistency by "snooping" on bus transactions. Shared memory and I/O are accessed through a common system bus.

Instruction Set Computer (RISC) [Patt85], and the second is a cache controller and bus interface chip. Also associated with each pair are a custom VLSI floating point co-processor and a collection of RAM and buffer chips (not shown). The shared global memory can be addressed up to 4 gigabytes.

The RISC processor is a tagged architecture with an instruction set tailored to execute LISP. It has a cycle time of about 150 nanoseconds. In the tradition of RISC, the Execution Unit contains a large register file, and control of the 32-bit datapath is handled by a four-stage pipeline [Kate83]. It is the task of the Instruction Unit to prefetch and buffer instructions in an effort to deliver one per cycle to the processor.

The caches in the system not only provide much faster access time than main memory, but also significantly reduce the total bus cycles needed for execution [Good83]. Without the caches, contention for the bus would limit the effective processors to just a few. Our studies have led us to specify each cache to be 128 kilobytes, direct-mapped, with transfers between memory and the cache handled in four-word (32 byte) blocks [Katz85a].

In any system with multiple caches, the problem of *cache consistency* arises. When a processor changes a block in its cache, other processors must not be allowed to read a "stale" copy residing in their own cache. To prevent this inconsistency, when cache writes occur copies of the same block in the other caches must either be updated or invalidated.

The caches in the SPUR system use a distributed *ownership* protocol to maintain consistency of shared data [Katz85b]. A block of memory may be contained in multiple caches for reading, but only one cache may "own" it for writing. Initially, all blocks are owned by memory. When a processor writes to a block not owned by its cache, the block's current owner relinquishes ownership and places the block on the bus. Any other caches with copies of that entry must invalidate it. To do this, the cache controller not only heeds processor requests, but is *dual-ported* to monitor bus requests: it "snoops" on the bus. By using a write-back policy, instead of write-through, bus traffic is even further reduced by only updating memory when a modified block must be flushed from the cache.

To provide minimal memory access time, SPUR uses *virtual address caches*: the caches are referenced directly by the processor with virtual addresses. Because of this, address translation is only required after a cache miss when a transaction to memory must be initiated. If the cache was addressed with physical addresses, the translation would have to occur on *every* processor reference. To retain reasonable effective memory access times, the translation mechanism would have to be extremely fast. Since cache misses only account for a small percentage of the total references, often around one or two percent, translation in SPUR does not have to be this fast to still yield high performance.

One appealing possibility is to design a system with virtual addresses on the bus. Translation would occur only when absolutely needed: at the memory. In an effort to minimize prototype design time, we decided not to modify the bus or memory devices, and this possibility was ruled out. The SPUR multiprocessor therefore performs translations at each processor, but only if the cache misses. See Appendix A for a more complete discussion of the considerations in locating address translation.

Two complications arise because of the decision to use virtual address caches, The first is the danger of ***synonyms***: two virtual addresses that refer to the same physical location. If this were allowed, it would be possible for two or more entries to appear in the cache for the same location in memory. This would complicate the task of maintaining cache consistency. In SPUR, synonyms are prohibited by the operating system.

The second problem is the need for ***reverse translation***: mapping a physical address back to virtual address. This is often done as a solution to the synonym problem, but arises in SPUR because of the need for the cache to snoop on the bus. Since the bus must transmit physical locations to memory, and the cache is referenced by virtual addresses, a reverse translation would have to be done. SPUR eliminates the need for reverse translations by transmitting both physical and virtual addresses over the bus. Caches snoop on the virtual address, and memory uses the physical address.

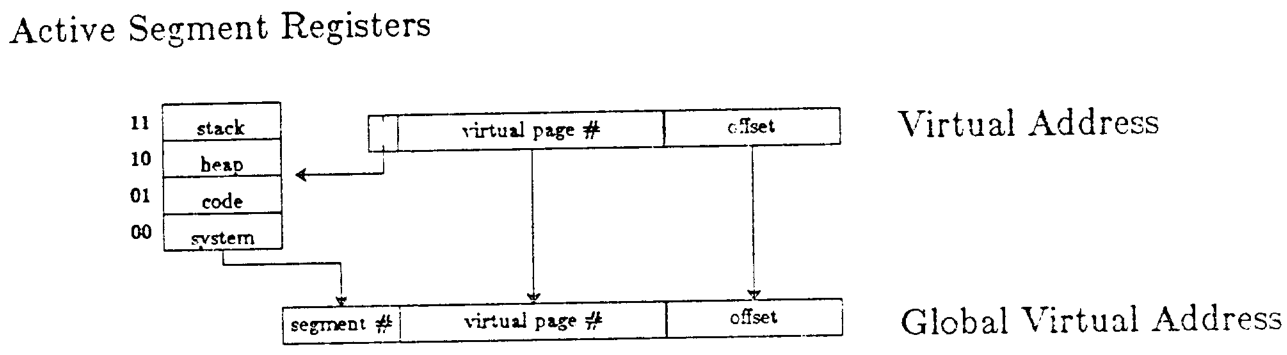


Figure 3.2 : Formation of a Global Virtual Address

The SPUR virtual memory allows for multiple large address spaces by providing one large global virtual address space. Each process's virtual address space is divided into four segments: stack, heap, code, and system. The global virtual address is formed in the cache by appending the segment number from one of the four active segment registers corresponding to these divisions.

The SPUR virtual memory model supports multiple address spaces by extending the processor's virtual address to a larger *global* virtual address. This global space is divided into 256 1-gigabyte segments, each mapped independently. As figure 3.2 shows, the top two bits of the processor virtual address are used to select one of four segments that are designated *active*. Thus, a process's virtual space is composed of four gigabytes divided into stack, heap, code, and system space. Processes share segments on an "all-or-nothing" basis: if any portion of a segment is shared by two processes, the whole segment must be shared.

Each segment is divided into 256K 4K byte pages. This implies that to map the entire global address space, over 64 million page table entries would have to

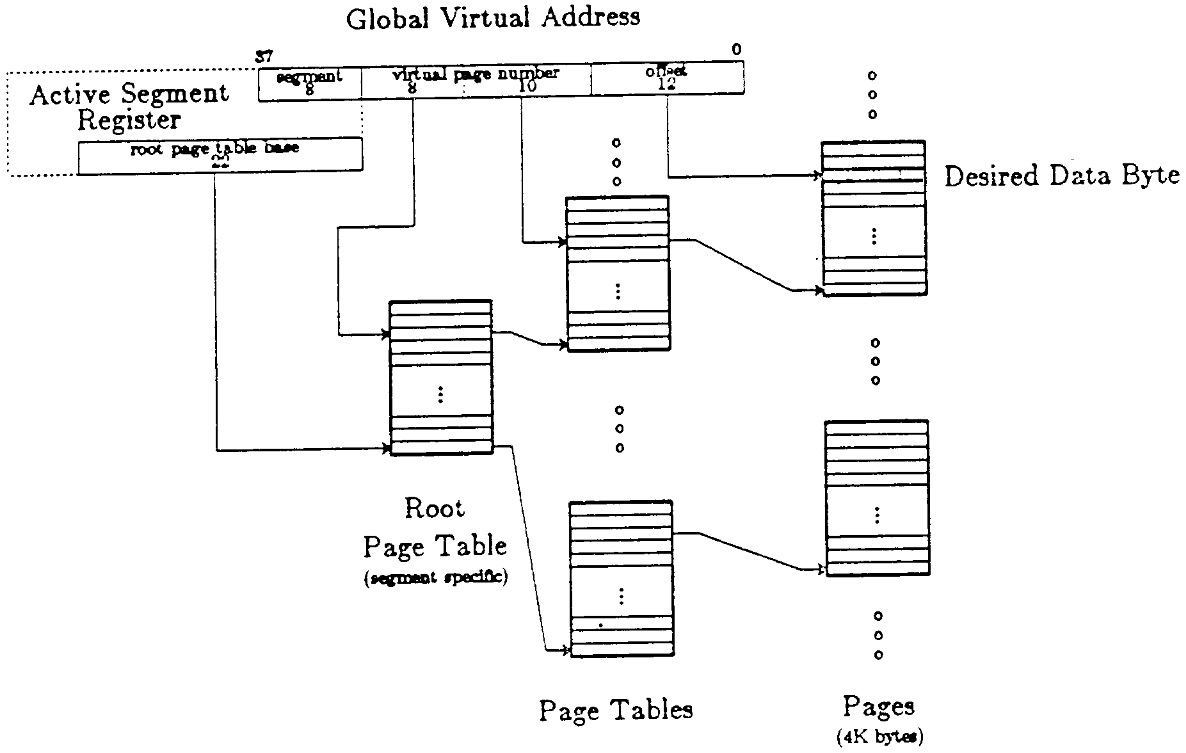


Figure 3.3 : Two-Level Page Tables for the Active Segment

Associated with the number of the active segment is the base address of the root page table for that segment. The high-order eight bits of the virtual page number index into the root page table to find the base of the appropriate page table. The low-order ten bits of the virtual page number select the page table entry for the desired page. The offset field then specifies the byte within the page. See Figure 3.5 for an explanation of how these addresses are formed in the cache controller.

be kept. Since each PTE is one word, this would require 256 megabytes of memory. By adopting a *two-level* page table structure, the page tables may to be written out to disk. Each of the "meta" page table entries, or *root* PTEs, maps one page of PTEs, for a total mapping 4 megabytes. Thus, these Root Page Tables require only 256K bytes to map the entire global address space, and are kept resident in memory. Figure 3.3 shows the two-level mapping structure for one segment.

3.2. The In-Cache Translation Process

Since translation is to be done at each processor, the same issue of data consistency that plagued multiple caches arises here as well. A translation buffer is really nothing more than a cache for page table entries, and snooping caches are designed to solve these consistency problems. Why not just use the existing caches to provide the translation? Aside from saving physical storage, there is now a second reason for the two-level structure that places the page tables in the virtual address space: PTEs must have virtual addresses in order to be cacheable. It has already been determined that the caches must be large to keep bus contention to an acceptable level, so it appears unlikely that the number of entries required for translation will cause much pollution. For example, the 128K byte cache holds 4,096 blocks. If only 32 of these happened to contain PTEs, this would be enough to map one megabyte of memory.

In SPUR the task of address translation is entirely the responsibility of the cache-controller chip. This device is a custom VLSI circuit and already requires a complex control for snooping and other operations like the writeback of dirty blocks and selective invalidation. The addition of control for address translation therefore represents only a small additional complication.

As shown before in Figure 3.2, the cache is referenced with the global virtual address formed by concatenating the selected active segment number to the virtual address supplied by the processor. Figure 3.4 shows the the four cases of operations that may be done to complete a memory reference.

In the most frequent case, the cache hits (A), and data is delivered in only one cycle. In preparation for a miss, a concatenate-and-extract circuit in the cache controller forms the virtual address of the page table entry during the reference. If translation is required, the cache controller uses this address and attempts to read the page table entry in the following cycle. Figure 3.5 shows how this address, and others in the translation process are formed.

Case B corresponds to a miss in the cache, and a hit in the "TLB". This requires one additional cache reference for the PTE, and one memory transfer to fetch the desired data block. If the PTE is not cached (C), the third cache reference is for the root PTE, from which the address of the PTE in memory may be formed. After the PTE is fetched from memory, it is loaded into the cache for use in future translations. In the worst case (D), all three cache references fail, and the root PTE must also be fetched from memory and cached. These root page

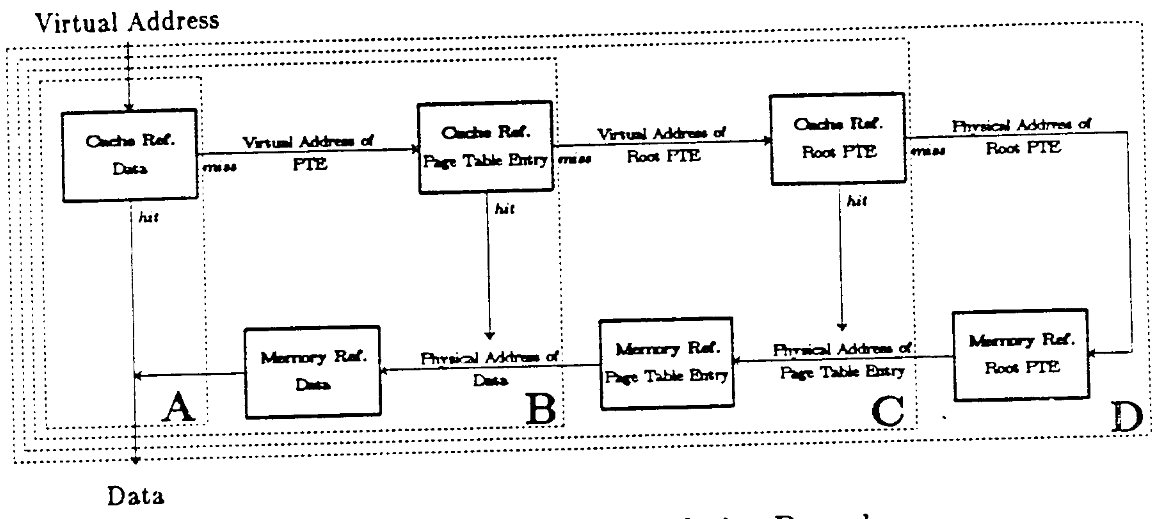


Figure 3.4 : Steps in the Translation Procedure

Four cases are possible depending on whether the cache contains the data, page table entry, and root PTE. A: The cache hits, and no translation is required. B: The first cache reference misses, but the cache contains the page table entry (a TLB hit). C: The second cache reference misses, but the cache contains the root PTE. D: The cache misses on all three attempts, and the root page table entry must be fetched from memory.

tables can always be found at physical locations associated with each of the four active segment numbers.

More than three memory operations may occur if a *write-back* of a cache block occurs. When a block from memory replaces a block in the cache that has been modified, this *dirty* block must be written back to memory. To perform the write without recursively needing another translation, the physical tag for each block is kept in cache tag-memory.

On examining any page table entry, the desired page may be shown to be *invalid*, indicating that the page is not in memory but resides on disk. In this event, a trap to the page fault handler is taken. The remaining bits of the PTE are used as an index into a table managed by the operating system that contains the disk addresses of the pages in secondary storage.

In most systems, *reference* and *dirty* bits for each page are kept in the PTE to handle the replacement and write-back of pages in memory. The SPUR system does not support true reference bits, but instead an approximation we refer to as the *miss* bit. A true reference bit would require bringing the corresponding page table entry into the cache for every reference to the cache. Instead, the miss bit

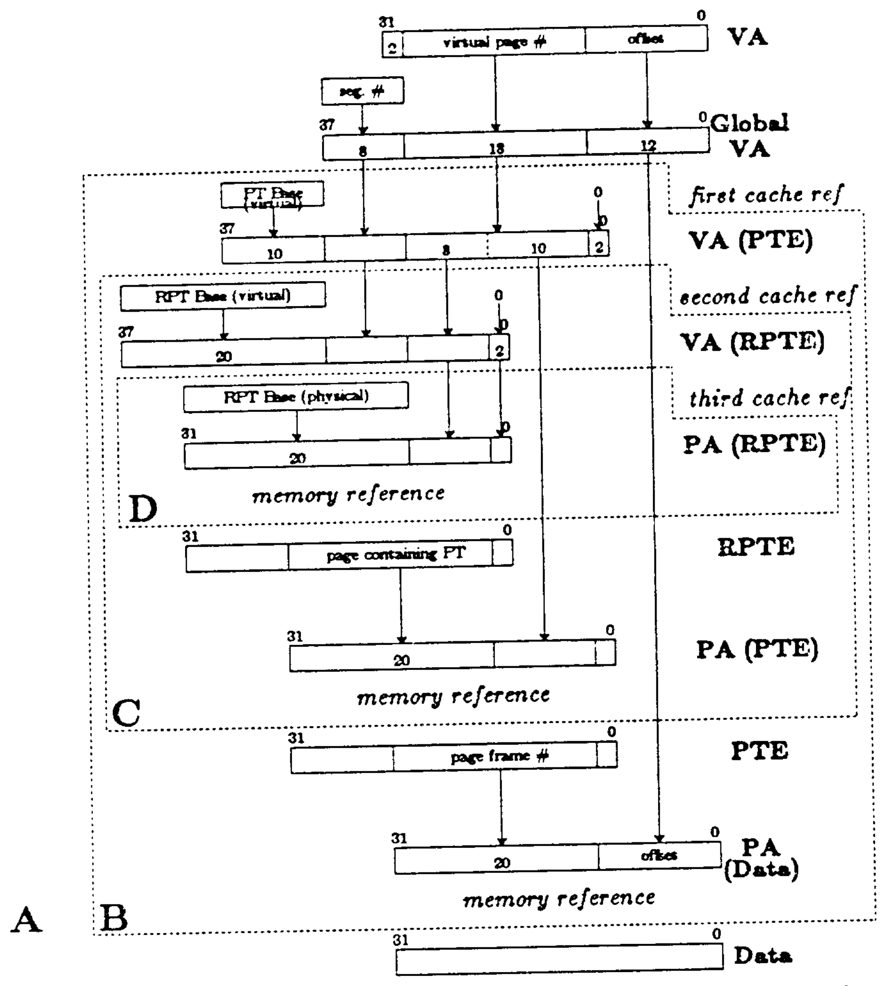


Figure 3.5 : Formation of Virtual and Physical Addresses in Translation

This figure shows how addresses are formed for the worst case scenario in the preceding figure. Up to three virtual addresses are formed to reference the cache for the requested data, page table entry, and root page table entry, respectively. At most, all three corresponding physical addresses must be formed and a separate bus-memory transaction is performed to fetch each block. The dashed lines divide the same four cases that were presented in Figure 3.4. The active segment number and page table base registers are contained in the cache controller.

is set only when a reference to a cache block misses. In this event, the PTE must be brought into the cache anyway to carry out the address translation. The operating system can periodically reset these bits at intervals observed to provide the best performance. The algorithm used for replacing pages in memory is therefore an approximation to a true "Least Recently Used" policy. In a similar attempt to limit the amount of writing to PTEs, the dirty bit is set not on every write to a block, but only when a cache requests write-ownership. A cache cannot write to a block until it has requested write-ownership, therefore every initial write by a cache will ensure that the dirty bit is set. Again, one chief advantage to this method of translation is that all information in the page table entries is kept consistent across the multiple processors.

4. Simulation of In-Cache Translation

4.1. Methodology

To evaluate the performance of the SPUR in-cache translation mechanism and other translation buffers, the DineroII cache simulator [Hill84] was used. This simulator is address-trace driven and reports miss rates and bus traffic for specified cache parameters. With modifications for the caching of page table entries, these simulations provided information about the cost of in-cache translation, and how it compares to using a separate translation buffer.

Address Traces Used

Trace	Description	Memory Referenced (Mbytes)	Memory Referenced (pages)
LISZT	Franz LISP self-compilation	0.6Mb	145
VAXIMA	Algebraic expert system (a derivative of MACSYMA)	1.7Mb	414
CS20K	Two VAXIMA streams interleaved every 20K references (Multi-user context switch rate)	2.5Mb	609
CS100K	Two VAXIMA streams interleaved every 100K references (Single-user context switch rate)	2.5Mb	609
MVS	Multiple calls to the MVS Operating system	3.7Mb	893

Table 4.1 : Address Traces Used

These five traces were used in the analysis of the SPUR in-cache translation scheme. The first four were generated on a VAX running UNIX. The last trace was recorded on an Amdahl 470 running the MVS operating system. The amount of virtual memory referenced is shown by the number of 4K byte pages that were touched.

Table 4.1 shows the five address traces used to drive the simulations and the amount of virtual memory that each references. The first four were gathered on a VAX running UNIX with an address and instruction tracer [Henr84]. LISZT is the Franz LISP compiler compiling itself. VAXIMA is an algebraic manipulator written in LISP performing a series of integrations, matrix operations, and solving differential equations. CS20K and CS100K are traces composed of two separate sections of the VAXIMA trace and designed to simulate context switching. They are identical except for the switching interval, which is 20,000 and 100,000 references, respectively. MVS is a series of calls to this operating system and was traced on an Amdahl 470 [Smit85]. This last trace references a much larger range of virtual memory.

Although the VAX instruction set is different from that of a RISC architecture, the only RISC traces available were of small program compilations that exhibited optimistic cache performance. The results of a study of the VAX TLB by Clark and Emer [Clar85] show miss rates higher than those produced by the RISC compilation trace, but were not as high as those of the VAXIMA trace (See Appendix B). SPUR is designed to be a symbolic machine, and VAXIMA is written is LISP, so this trace is more typical of programs that will be run. The behavior of VAXIMA is therefore a conservative estimate of TLB performance.

Measurements of timesharing systems like the VAX [Emer84] show context switches occurring about every 6500 instructions. This corresponds to roughly every 20,000 references including both instructions and data. There is less experience with single-user machines, but their interrupt rates should be dominated by the pace of one person interacting with the workstation. This would suggest context switch rates of about 10 to 100 a second. With a cycle time approaching 100 nanoseconds, the 100 interrupts per second would result in switching every 100,000 references.

The traces CS20K and CS100K were used to simulate context switching. They interleave two different Vaxima traces at intervals of 20,000 and 100,000 references, respectively. Rather than flush the cache on context switch, the two reference streams are in separate address spaces. The SPUR caches will not need to be flushed on context switch. Although blocks from different contexts can displace each other in the cache, the use of segment numbers will ensure different virtual addresses and will therefore not cause false hits. On a multiprocessor designed for one user, the number of separate processes active on one processor is likely to be small. Hence, only the two streams are interleaved.

The instruction and address tracer used on the VAX is capable of measuring user processes only. There is therefore nothing to show system performance, and the MVS trace was acquired for this purpose. This particular section of the trace shows extremely poor locality and as Table 4.1 shows, references a much larger range of virtual memory than the other traces. There is a high frequency of MVS memory references whose addresses agree in the low-order bits, causing them to index to the same entries in a cache. MVS therefore shows unusually high rate of cache collisions: cases where one reference "bumps out" an older block being stored. Over 12.5% of all references are to just the first 32 byte block. Collisions here account for 15.3% of all misses when simulating a direct-mapped cache. MVS then, provides a solid upper bound on miss rates, and accentuates the characteristics of the cache when varying parameters (see Appendix C).

All five traces contain one million references. Although this represents under one second of execution, this length was necessary given the available resources. Several longer traces of five million references were run and miss rates did not differ to within one-hundredth of a percent.

4.2. Performance of In-Cache Translation

There are two opposing views of the SPUR cache/translation buffer: a cache being corrupted by page table entries, or a translation buffer being polluted by instructions and data. Table 4.2 shows the result: an increase in cache miss rate because both functions are being performed in the cache. This total additional miss rate is computed by dividing the misses added when PTEs are cached by the total number of references made to the cache *by the processor*.

There is an important distinction to be made: *processor* references to the cache are for instructions and data, while the *cache* refers to itself for page table

Increase in Cache Miss Rate

Trace	Miss Rate (%) Pure Cache	Miss Rate (%) w/Translation	Additional Cache Misses Total	(Collisions)	(PTE Misses)
LISZT	0.584	0.609	0.025(4.3%)	(0.009)	(0.016)
VAXIMA	1.855	1.885	0.030(1.6%)	(0.004)	(0.026)
CS100K	2.214	2.260	0.046(2.1%)	(0.005)	(0.041)
CS20K	2.445	2.494	0.049(2.0%)	(0.007)	(0.042)
MVS	8.740	12.122	3.382(38.7%)	(0.994)	(2.388)

Table 4.2 : Additional Cache Misses Due To In-Cache Translation

This table shows the increase in cache misses when translation is performed in-cache. For example, the miss rate for VAXIMA is 1.855% when PTEs are not cached for translation. The additional 0.03% is composed of two elements: extra misses when the processor references instruction and data blocks that have collided with PTEs (0.009%), and the misses when the cache references itself for a page table entry (0.016%).

entries in the translation process. The last two columns of Table 4.2 separate the total additional miss rate according to this distinction. In the column labelled "Collisions", the processor is experiencing additional misses on instructions and data because normal cache contents are being displaced by PTEs. This is strictly a *cost* of performing translation in the cache. The "PTE Misses" column, on the other hand, reflects the additional misses incurred only when the cache is being referenced for the translation process. This is the measure of the *performance* of the translation mechanism: the "TLB" miss rate for SPUR. This measure includes not only the misses on page table references, but root page tables as well. As we shall see, there are few root page table references.

Table 4.3 displays the percent of memory references handled by each of the four cases shown in Figure 3.4. Between 90 and 99% of the time, the cache hits, and reference is handled in one cycle (A). The SPUR in-cache translation takes over on a miss, and in the next cycle references itself with the virtual address of the page table entry. From 0.5% to 7% of all references are cache hits on these references (B). The desired instruction or data can then be fetched from memory in one bus transaction.

For all the traces except MVS, only 2 or 3 references out of 10,000 miss in the "TLB" and thus go to memory for the page table entry (C). Only about 2 in 100,000 references take a "double-miss," and require memory fetch of the root page table entry as well (D). These second level lookups represent only 3.3% of PTE misses on average. This agrees with what Clark and Emer report to be 3.1 to 4.8% [Clar85] and supports the use of a two level page table scheme.

Types of Memory Accesses

Trace	Percentage of Total References				Average Access Time (cycles)
	Cache Hits A (1$)	PTE Hits B (2$,1M)	RPTE Hits C (3$,2M)	RPTE Misses D (3$,3M)	
LISZT	99.4065	0.5775	0.0158	0.0002	1.085
VAXIMA	98.1408	1.8345	0.0231	0.0016	1.264
CS100K	97.7806	2.1805	0.0373	0.0016	1.316
CS20K	97.5478	2.4115	0.0390	0.0017	1.349
MVS	90.2655	7.3540	2.3732	0.0073	2.697

Table 4.3 : Breakdown of Memory References

This table shows the percentage of total references to memory that fall into the four categories first shown in Figure 3.4. The average number of cycles per reference is given for each trace. A "$" represents a cache reference 1 cycle, and an "M" indicates a memory transaction requiring 13 cycles.

4.3. Comparison to a Separate TLB

Table 4.4 shows how the SPUR in-cache translation compares to the commercial translation buffers that were presented in Table 2.1. The SPUR method of translation displays consistently lower miss rates for all but one case (LISZT on the 470V/8). The more poorly-behaved the trace, the better the in-cache method does when compared to the commercial buffers. By allowing a large, variable number of entries to be dedicated to translation, the SPUR scheme substantially outperforms even the large, set-associative buffers that use hashing.

A good deal of the high performance displayed by the SPUR system could be because the translation is only being done on cache miss. All the commercial systems shown in the table translate on every reference. The following more direct comparison accounts for the presence of the large virtual cache.

Table 4.5 shows the performance of using a separate translation buffer placed after the SPUR cache. These figures were generated by simulating the performance of the 128K byte, direct-mapped cache for each trace, and recording only those addresses that missed. These were then used as the input to each of the simulated translation buffers.

Summary of Commercial TLB Performance

Machine	Miss Rate (Percent)				
	LISZT	VAXIMA	CS100K	CS20K	MVS
VAX-11/730	3.588	4.070	4.332	4.444	13.109
VAX-11/750	1.782	2.306	2.344	2.460	11.226
VAX 8600	0.639	1.249	1.545	1.710	10.277
VAX-11/780	0.324	0.619	0.676	0.856	8.913
IBM 370 3033	0.097	0.305	0.450	0.550	6.401
Amdahl 470V/6	0.023	0.112	0.174	0.223	5.137
Amdahl 470V/8	**0.015**	0.047	0.086	0:101	3.336
SPUR In-Cache	0.016	0.026	0.041	0.042	2.388

Table 4.4 : Commercial TLB Performance

Simulations of the VAX TLBs are for one half only. Recall that one half is available to user programs while the other half is reserved for system space translations. The IBM and Amdahl performance were simulated using a hashed index based on an Exclusive OR of the address bits. By allowing a large, variable number of entries to be dedicated to translation, the SPUR in-cache method displays consistently lower miss rates than all of these buffers except for LISZT (in bold font). The figures for SPUR are from Table 4.2.

Trace	Separate TLB Performance								SPUR In-Cache	
	Miss Rate (%)								Miss Rate (%)	Average Number of PTEs
	Direct-Mapped				8-way Set-Associative					
	128	256	512	1024	128	256	512	1024		
LISZT	0.072	0.023	**0.003**	**0.003**	0.049	**0.003**	**0.003**	**0.003**	0.016	98
VAXIMA	0.395	0.232	0.162	**0.010**	0.258	0.126	0.033	**0.010**	0.026	294
CS100K	0.492	0.303	0.229	0.044	0.297	0.144	0.065	**0.027**	0.041	457
MVS	5.570	4.673	3.988	3.059	4.776	4.345	3.620	2.535	2.388	920

Table 4.5 : Performance of Separate TLB After Cache Miss

This table shows separate TLBs which are placed after the SPUR virtual cache. Bold font indicates better performance than the in-cache method. These cases require over twice the number of entries used by the SPUR translation since the separate TLBs are of fixed size.

To get miss rates as low as those shown in Table 4.2, a separate TLB would require over 256 entries for LISZT, 512 entries for VAXIMA, and over 1024 entries for the context switching traces and MVS. Even if the buffer were built to be 8-way set-associative, to do as well would require over 128, 512, and 1024 entries, respectively.

The last column in Table 4.5 shows the average number of PTEs contained in the SPUR cache using in-cache translation. Since the separate TLBs are of fixed size, they require over twice the entries to do as well. The direct-mapped TLBs would need over three times as many entries. Although instructions and data can displace page table entries in the in-cache scheme, having a large *variable* number of entries appears to overshadow this.

Table 4.6 shows the commercial translation buffers examined before in Table 4.4, but this time translating only when the SPUR cache misses. The "Cache Miss" columns have identical entries because the SPUR 128KB cache was simulated in each case. The cycles required for the average reference were calulated as in Table 4.3. The relative cost when compared with the cycles required for SPUR is displayed in the last column for both VAXIMA and MVS.

The SPUR in-cache translation had lower TLB miss rates than all the buffers except for the Amdahl 470 V/8 when running MVS. The V/6 and V/8 both do better than SPUR when taking into account the total machine cycles required. This is because of the high occurrence of PTE misses we observed in Table 4.2 for MVS. Note that in Table 4.4 these separate translation buffers showed a steady

Commercial TLB Performance with SPUR Virtual Cache								
	VAXIMA				MVS			
Separate TLB	Cache Miss (%)	TLB Miss (%)	Avg. Time (cycles)	Rel. Cost (/SPUR)	Cache Miss (%)	TLB Miss (%)	Avg. Time (cycles)	Rel. Cost (/SPUR)
64 PTEs, 1-way (VAX-11/730)	1.855	0.368	1.311	1.038	8.740	6.579	3.145	1.166
64 PTEs, 2-way (VAX-11/750)	1.855	0.308	1.303	1.032	8.740	5.805	3.036	1.126
256 PTEs, 1-way (VAX 8600)	1.855	0.130	1.278	1.012	8.740	4.408	2.841	1.053
256 PTEs, 2-way (VAX-11/780)	1.855	0.088	1.272	1.007	8.740	4.222	2.815	1.043
128 PTEs, 2-way (IBM 370 3033)	1.855	0.158	1.281	1.014	8.740	4.064	2.792	1.035
256 PTEs, 2-way (Amdahl 470V/6)	1.855	0.081	1.271	1.006	8.740	3.179	**2.669**	**0.990**
512 PTEs, 2-way (Amdahl 470V/8)	1.855	0.046	1.266	1.002	8.740	**1.941**	**2.495**	**0.925**
SPUR In-Cache	1.859	0.016	1.263	1.000	9.734	2.387	2.697	1.000

Table 4.6 : Performance of Commercial TLBs After Cache Miss

These are the same commercial translation buffers as in Table 4.4. Here, however, they are placed after the SPUR virtual cache to show performance when translating only on cache misses. TLB miss rate and cycles required were lower for the SPUR in-cache translation method for all except the largest TLBs with MVS (shown in bold). As before, only half the entries for the VAX buffers were simulated and the IBM and Amdahl TLBs use a hashed index.

decrease in miss rate from the VAX 11/780 to the Amdahl 470 V/8. The same is not true for their performance after a virtual cache. Here, the 256 entry VAX buffers do as well or better than the two directly below them that use hashing. This suggests that hashing is less important for translation after a cache. The number of entries in the device is clearly the dominating factor.

5. Conclusions

In SPUR, the desire for single-cycle cache access dictates that caches be virtually addressed. Without modifications to the bus and memory, options for performing address translation at main memory are ruled out. This means translation must occur after the cache and before the bus. For the large virtual address space and physical memory of the workstation, a translation buffer provides the most effective use of a "small" amount of mapping memory and a two-level page table scheme reduces the size of the full map in physical memory. Placing the page tables in the virtual address space allows the page table entries to be cached.

In-cache translation solves the problem of data consistency between multiple translation devices by using the cache's snooping protocol, thus avoiding additional hardware at the cost of complicating the cache-controller. Since the SPUR multiprocessor allows shared data to be cached, the cache controllers must already be complex enough to support a cache-consistency protocol.

Translation buffers have been separate devices for purely historical reasons. Traditionally, translation has occurred before (or in parallel with) referencing the cache. With a virtual cache, there is no reason that the existing hardware cannot be used for both purposes. Since a small number of cache entries are needed to hold page table entries, the cache performs address translations with minimal effect on the normal performance.

The increase in cache miss rate due to in-cache translation has two components: references by the processor that miss because of added collisions due to PTEs, and references to PTEs by the cache that miss. Of these, the occurrence of PTE misses is much larger than the amount of instructions and data being displaced. When this PTE miss rate is compared against existing TLBs, it outperforms even large, set-associative buffers using hashed indexing. If a separate translation buffer were used, it would need to be over twice the size of the number of entries required on average by the in-cache method. Even if it were made highly associative, this would still demand 512 or more entries to do as well. A larger, variable number of entries outweighs the cost of additional cache misses incurred. See Appendix C for the effect of a different cache organizations on the SPUR in-cache method of translation.

Studying this form of translation was the result of the particular requirements of the SPUR multiprocessor. These results hold for uniprocessors as well. Perhaps a significant effect of this work will be in the area of low-cost computers. In the past, personal computers have rarely had cache memory because of the cost-performance trade off. With the decline in memory costs, more small systems will begin to feature larger caches. VLSI now makes it possible to build a circuit with controller and cache tags on-chip. In effect, adding the control for address translation yields a TLB for free.

6. Acknowledgements:

A number of people working on the SPUR project helped to make this research possible. The DineroII cache simulator was written by Mark Hill and was later modified by David Wood to measure in-cache translation. George Taylor and Robert Henry were responsible for gathering the VAX traces. Alan

Smith provided me with the MVS trace. Professors David Patterson, John Ousterhout, and Randy Katz deserve special note for their advice and patience. Garth Gibson, Susan Eggers, Paul Hansen, Joan Pendleton, and Brent Welch also provided me with ideas and the necessary supportive harassment. In general, my thanks go to all the faculty and graduate students involved in the SPUR research group.

Appendix A
Issues in the Location of Address Translation

In deciding how to support virtual memory, the issue arises of precisely *where* the use of virtual addresses ends and physical addresses begin. Figure A.1 identifies five potential locations for address translation in the SPUR multiprocessor:

(1) Before the Instruction Buffer (IB)
(2) Between the Instruction Buffer and the cache
(3) In parallel with the cache
(4) Between the cache and the system bus
(5) Between the system bus and memory

Placing the translation before the Instruction Buffer (IB) (option (1)) has significant disadvantages. Address translation is required for every reference, reducing the advantage of having the instruction memory on-chip. The IB is likely to be small enough so that the low-order bits of the virtual address can be used to directly access the buffer. Recall that the low-order bits represent the byte within a page and therefore are not effected by translation. Since physical addresses are not needed at this point, we cannot justify dedicating processor chip area to translation hardware.

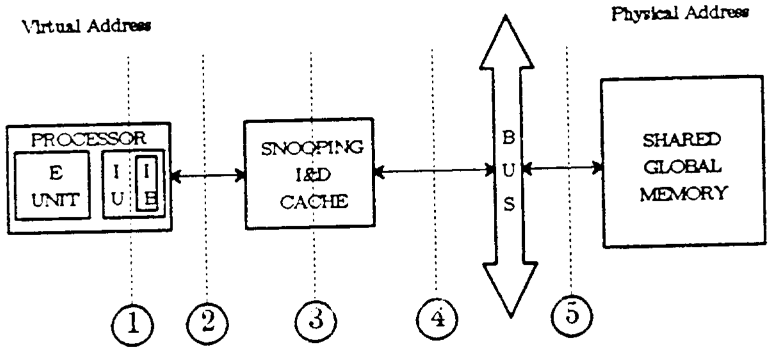


Figure A.1 : Potential Locations for Address Translation

The potential locations for address translation are shown: (1) before the instruction buffer (IB), (2) between the processor and the cache, (3) in parallel with the cache, (4) between the cache and the bus, and (5) between the bus and memory. Note that for simplicity only one of multiple processor-cache pairs is shown.

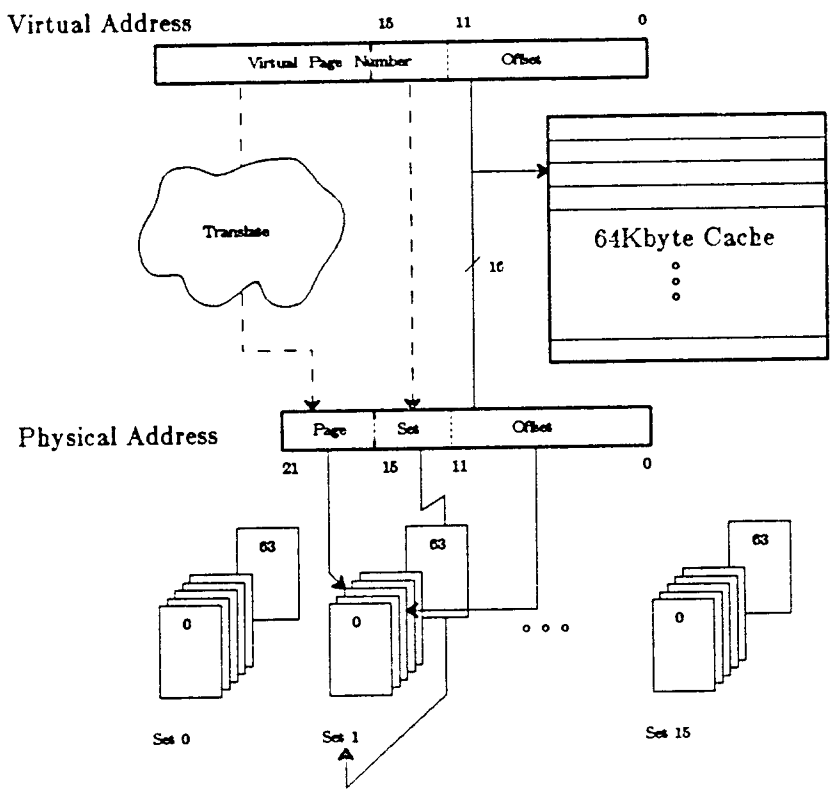
Alternatively, the translation can be done between the IB and the cache (option (2)). This is the approach taken in the VAX machines [DEC 81], and allows the IB to be accessed without translation. However, data references and IB misses still incur the overhead of translation. Both this and the preceding option require doing the translation in series with the cache access. To keep the time for a cache reference to a minimum, we reject both alternatives.

Rather than doing the translation in series with a cache access, the third alternative is to do it in parallel (option (3)). Such a scheme reduces the cost of a cache reference, potentially to a single RAM cycle, since it is now not the sum, but the *maximum*, of translation and cache access times. The cache tag memory and the translation buffer are accessed with the virtual address in parallel. The resulting tag and physical page number are then compared to determine if the cache has a hit. This is the approach taken on many of the IBM 370 mainframes [Smit82].

Since the cache is now referenced from the processor with *virtual* addresses, two complications are introduced: synonyms and the need for reverse translations. *Synonyms* arise when more than one virtual address refers to the same physical address. This complicates cache consistency because there is no longer a straightforward mapping between virtual and physical addresses. For example, it is possible to have multiple copies of the same memory block stored in a cache with different virtual addresses. *Reverse translation* occurs when a physical address on the bus must be translated back into a virtual address. This is often done as a solution to the synonym problem, but arises in SPUR because of the need for the cache to snoop on the bus. Since the bus must transmit physical locations to memory, and the cache is referenced by virtual addresses, a reverse translation would have to be performed. Special mechanisms are required to do this mapping, such as reverse translation buffers or a fully-associative organization for the cache tags.

It should be noted that if the size of the cache is small enough, only bits from the "byte-on-page" field are needed to identify a set. Since this field does not change during translation, there is no need for reverse translation. Increasing the associativity of the cache has the effect of reducing the number of sets for the same amount of cache. This explains why it might be advantageous to build caches with set-associativity of 16 or more even though studies have shown that 8-way set-associativity well approximates full associativity [Smit78].

Even if the cache must be addressed with bits from the page-number field, reverse translation could still be avoided by requiring that physical page numbers match their corresponding virtual page numbers in enough bits. For example, if only one additional bit beyond the byte-on-page field were required to address the cache, we could arbitrarily require that even numbered virtual pages be placed only in even numbered physical page frames, and similarly odd pages could only be mapped to odd page frames. In general, a virtual page would be restricted to reside in a particular *set* of physical page frames. Figure A.2 shows a more realistic example of this *set-associative page placement*. The cache may now be referenced from either the virtual or physical "side" with no need for a reverse mapping. However, if programs reference only certain sets heavily, unnecessary paging may occur even though there may be available frames in other sets.



Page Frames in a 4Mbyte Memory

Figure A.2 : Set-Associative Page Placement in Main Memory

If virtual and physical addresses are constrained to match on enough of the low-order bits, it becomes possible to map either address into the same cache locations. However, this restricts a physical page frame to hold only virtual pages whose addresses coincide on these low order bits. For example, a 64K byte (16 address bits) direct-mapped cache requires that the low order 16 bits of the physical and virtual address match. Assuming a 4MB main memory (22 address bits) with 4K byte pages, virtual pages can be placed into one of 16 sets (selected by address<15:12>) of 64 physical pages (selected by address<21:16>). This means that there are only 64 possible page frames for each virtual page, rather than 1024 (the total number of physical page frames).

The need for reverse translations can be avoided more simply by placing both virtual and physical addresses on the bus. The caches are addressed from the system bus side by virtual addresses; physical addresses are used to access main memory. This requires either a wider address bus or time multiplexing the addresses.

In the preceding three options, the translation must be done on every cache reference. For high performance, this requires that the translation mechanism be fast. A good deal of hardware and design effort must be spent to keep the mapping time down to one RAM access.

The fourth alternative translates only on a cache miss (option (4)). This is attractive since misses constitute a small percentage of all references. Thus, a slower mechanism built with less hardware can achieve the same effective access time as the previous, more costly mechanisms. The need for reverse translations is still present and requires the same mechanisms as discussed for option (3).

The Xerox Dragon [McCr84], a VLSI-based multiprocessor system with a similar architecture as that described here, does its address translations only if a cache misses. Reverse translation is handled by storing both physical and virtual page numbers for each block in the tag memory, and by providing a fully associative lookup from the system bus side. Even on a miss, a translation is not necessarily required: if the referenced word is on the same virtual page as some other word already in the cache, translation is avoided by using the physical page number stored with that word's block. This is an elegant solution, but requires a fully associative lookup and roughly twice the memory for cache tags. Both of these costs severely limit the amount of cache that can be provided on a single VLSI chip.

The final option is to do the address translation in the main memory system (option (5)); the system bus would then use only virtual addresses. This has the advantage of centralizing the mapping hardware. The contention for this hardware would be no worse than that of main memory itself. There are, however, several disadvantages. First, the bus must be wider than a strictly physical bus to accommodate the larger virtual address. Second, latency to memory must increase to allow for translation. For protocols in which the bus is "held," the bus will be busy for a longer period of time per reference. Since the bus is a critical resource in a tightly coupled multiprocessor, this is likely to have a serious effect on performance. It might be possible to design the translation mechanism to work largely in parallel with the memory RAM access. However, we lose the advantage of being able to do it in the "leisurely" fashion discussed in option (3). By translating at the memory, reverse translations are not needed, but synonyms could still present a problem. To simplify the cache consistency protocol, writable synonyms would have to be disallowed. This approach also requires the design of custom memory and I/O controllers.

For high performance in the SPUR multiprocessor, we chose to provide a virtual cache, and to do translation only on misses. This rules out the first three options. Prototyping constraints prevented the redesign of memory and I/O eliminating the last option. Translation is therefore performed at each processor after the cache. Both physical and virtual addresses are placed on the bus to eliminate the need for reverse translation, and synonyms are disallowed by requiring that two address spaces sharing common data use the same virtual addresses for that region.

Appendix B
The Effect of Page Size on the VAX TLB Miss Rate

Simulations of the VAX-11/780 translation buffer show that performance would be significantly improved by increasing the page size from 512 to 4K bytes. The results of my simulations and those of Clark and Emer [Clar85] are shown in Figure B.1. The figures shown by Clark and Emer closely coincide with the values produced when using the traces RISC.RCOM.PUZZLE, SPICE, and VAX.RCOM.PUZZLE. The curve for the VAX.RCOM.PUZZLE trace shows that

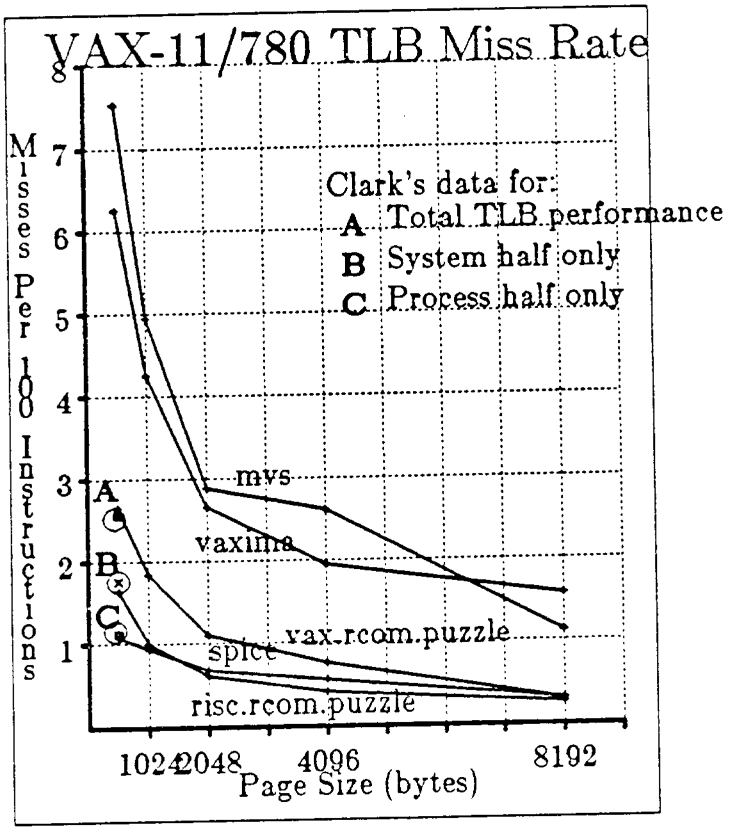


Figure B.1 : Effect of Page Size on the VAX TLB Miss Rate

This graph shows the advantage of larger page size in translation. Misses in the VAX translation buffer would drop from about 2.5 in 100 instructions down to 0.5 in 100 if the page size were increased from 512 to 4K bytes. The VAXIMA and MVS traces are used in the study of the SPUR in-cache translation, and show much higher miss rates.

misses in the VAX translation buffer would drop from about 2.5 in 100 instructions down to 0.5 in 100 if the page size were increased from 512 to 4K bytes.

These simulations were performed with one half of the VAX translation buffer. The curve for MVS was generated without flushing to approximate system half performance. The other curves are the result of flushing the process half every 20,000 references.

The VAXIMA and MVS traces are used in the study of the SPUR in-cache translation, and show much higher miss rates. These results further strengthen the assumption that the VAXIMA trace produces high enough miss rates to serve as a conservative measure of translation buffer performance.

Appendix C
Sensitivity of In-Cache Translation to Cache Parameters

In the following studies, the parameters of the SPUR cache were used as the nominal values: 128K byte unified cache, direct-mapped, 32 byte block size, with a 4K byte page. Cache size, associativity, block size, and page size were then varied to examine the effect on the SPUR in-cache translation method. In the following graphs, the increase in percentage miss rate is plotted on a log scale on the vertical axis. This metric reflects both the collisions due to the presence of PTEs in the cache, and also the misses on the PTEs themselves.

Figure C.1 shows that even sizes that are small relative to the 128Kb SPUR cache, the translation performs well when compared with commercial translation buffers. For example, LISZT in a 16Kb cache incurs an increase in miss rate from 2.7% as a normal cache, to 3.2% with in-cache translation: an additional 0.5% misses. This increase in miss rate roughly halves when these smaller sizes are

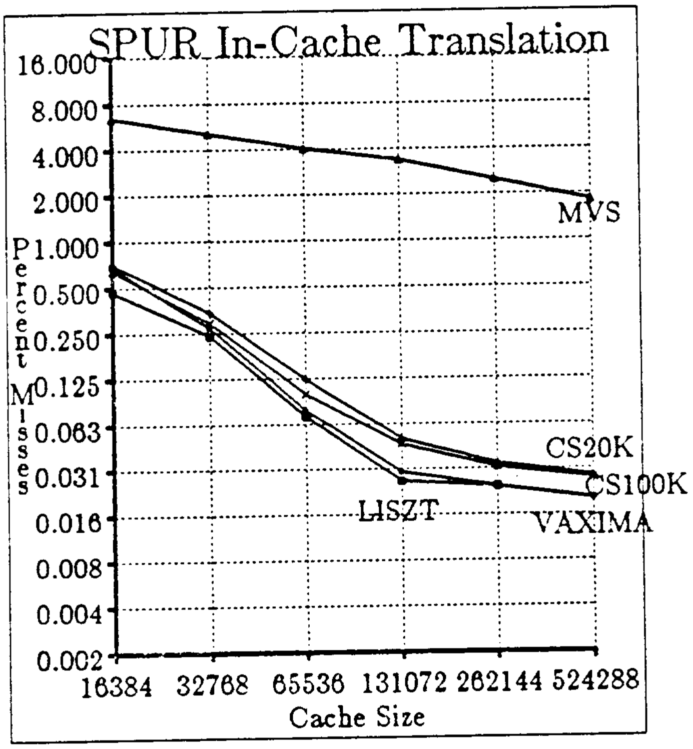


Figure C.1 : Increase in Miss Rate over Variations in Cache Size

increased by a factor of two. The "knee" of the curve at 128K bytes is typical of observations that led to building the cache at this size.

In Figure C.2, both LISZT and MVS show considerable sensitivity to associativity. The two context-switching traces, CS100K and CS20K display what appears at first to be odd behavior. At 2-way set-associativity, they do better than direct-mapped, but as the cache approaches full associativity, it begins to cost more to do translation in-cache. This is due to the rate at which one reference stream collides with entries left over from the other stream's previous run. In set-associative caches, all entries are not replaced until every entry in each set has been indexed to. However, in a fully-associative cache, all **n** entries will be replaced as soon as **n** new blocks have been brought in. The graph argues strongly for increasing the associativity to two-way. However, by building the cache as direct-mapped, circuit area that would have been used for multiplexing hardware was traded for more tag storage.

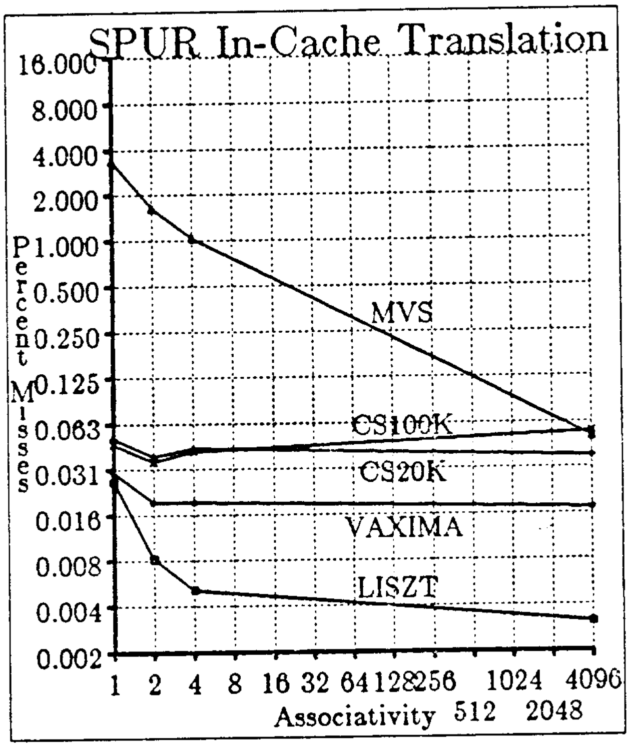


Figure C.2 : Increase in Miss Rate over Variations in Associativity

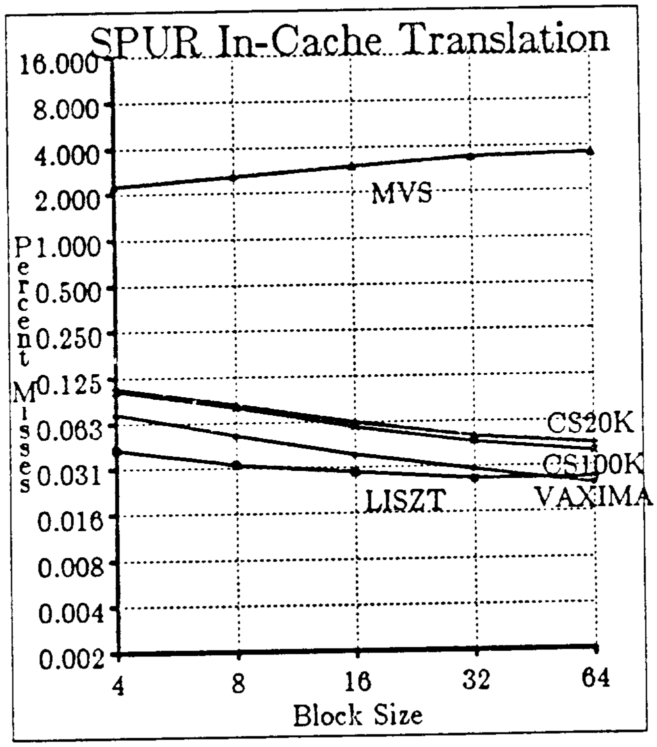


Figure C.3 : Increase in Miss Rate over Variations in Block Size

Figure C.3 displays much less sensitivity to variation than the preceding graphs. For MVS, the additional misses increase as the block size increases. On closer inspection, this is most dependent on the occurrence of PTE misses. The cases of PTEs colliding with instructions and data reach a slight minimum at 32 byte blocks. For the other four traces, both of these values steadily decline as block size increases. This suggests that MVS references adjacent pages less frequently.

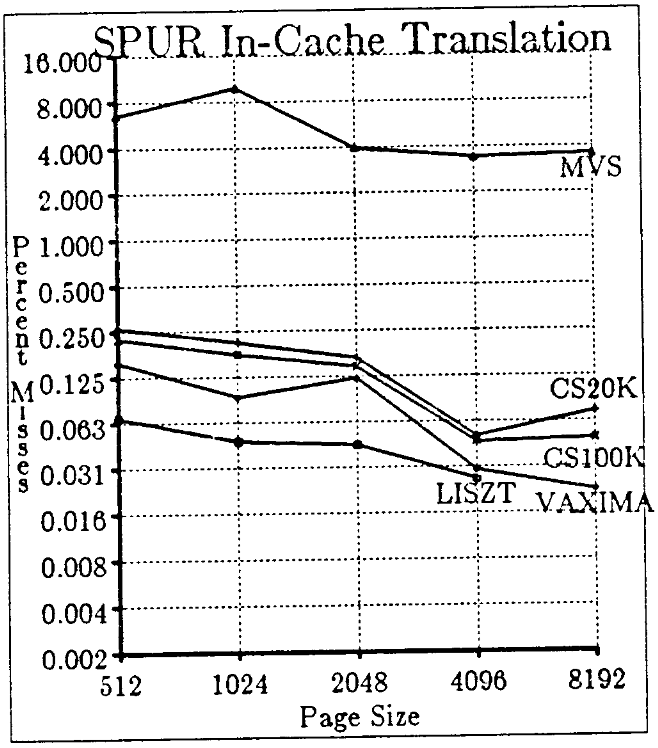


Figure C.4 : Increase in Miss Rate over Variations in Page Size

Figure C.4 also shows less sensitivity to variation than cache size or associativity. In general, as the page size increases, the translation is more effective. This is almost entirely accounted for by the number of PTE misses declining. The number of collisions of PTEs with instructions and data also drops, but to much less of an extent. For some reason, a page size of 2K bytes causes a higher rate of these collisions in the Vaxima trace.

References

[Bech82] Bechtolsheim, Andreas, Forest Baskett, Vaughn Pratt, "The SUN Workstation Architecture." Stanford University, Computer Systems Lab, Stanford California, CSL-TR-229, 1982.

[Bens72] Bensoussan, A., C.T. Clingen, and R.C. Daley, "The MULTICS Virtual Memory: Concepts and Design." *Communications of the ACM,* Vol. 15, No. 5, May 1972, pp. 308-318.

[Case78] Case, Richard P., Andris Padegs, "Architecture of the IBM System/370." *Communications of the ACM,* Vol. 21, No. 1, January 1978, pp. 73-96.

[Clar81] Clark, D.W., B.W. Lampson, K.A. Pier, "The Memory System of a High-Performance Personal Computer." Xerox Palo Alto Research Center, Technical Report CSL-81-1, January, 1981.

[Clar85] Clark, D. W., J. S. Emer, "Performance of the VAX-11/780 Translation Buffer: Simulation and Measurement." *Transactions on Computer Systems,* Vol. 3, No. 1, February, 1985.

[Dale68] Daley, Robert C., Jack B. Dennis, "Virtual Memory, Processes, and Sharing in MULTICS." *Communications of the ACM,* Vol. 11, No. 5, May 1968, pp. 306-312.

[DEC 81] Digital Equipment Corporation, *VAX Architecture Handbook,* DEC Sales Support Literature Group, Maynard, Mass., 1981.

[Denn72] Denning, P. J., "On Modeling Program Behavior." Proc. Spring Joint Computer Conference, AFIPS Press, Arlington, Va, Vol. 40, 1972.

[Emer84] Emer, J. S., D. W. Clark, "A Characterization of Processor Performance in the VAX-11/780." 11th Annual Symposium on Computer Architecture, Ann Arbor, MI, June 1984.

[Foth61] Fotheringham, John, "Dynamic Storage Allocation in the Atlas Computer, Including an Automatic Use of Backing Store." *Communications of the ACM,* Vol. 4, No. 10, October 1961, pp. 435-436.

[Good83] Goodman, J., "Using Cache Memories to Reduce Processor--Memory Traffic." Proc. 10th Annual Symposium on Computer Architecture, Stockholm, Sweden, June 1983.

[Henr84] Henry, Robert R., "Address and Instruction Tracing for the VAX

Architecture." University of California, Berkeley, November, 1984.

[Hill84] Hill, M. D., A. J. Smith, "Experimental Evaluation of On-chip Microprocessor Cache Memories." 11th Annual Symposium on Computer Architecture, Ann Arbor, MI, June 1984.

[Kate83] Katevenis, M. G. H., "Reduced Instruction Set Computer Architectures for VLSI." UC Berkeley, Computer Science Division, Technical Report UCB/CSD 83/141, October 1983.

[Katz85a] Katz, R. H., S.J. Eggers, G.A. Gibson, et al., "Multiprocessor RISCs: Initial Analyses and Results." UC Berkeley, Computer Science Division, Technical Report UCB/CSD 85/221, January 1985.

[Katz85b] Katz, R. H., S. Eggers, D. Wood, C. Perkins, R. Sheldon, "Implementing a Cache Consistency Protocol." To appear, Proceedings 12th Annual Symposium on Computer Architecture, Boston, MA, June 1985.

[Pier83] Pier, Kenneth A., "A Retrospective on the Dorado, A High-Performance Personal Computer." Xerox Palo Alto Research Center, Technical Report ISL-83-1, August, 1983.

[McCr84] McCreight, Edward. M., "The DRAGON Computer System: An Early Overview." NATO Advanced Study Institute on Microarchitecture of VLSI Computers, Urbino, Italy, July 1984.

[Patt85] Patterson, D., "Reduced Instruction Set Computers." *Communications of the ACM,* Vol. 28, No. 1, January 1985.

[Stre78] Strecker, W.D., "VAX-11/780 -- A Virtual Address Extension to the DEC PDP-11 Family." *AFIPS Proceedings NCC,* 1978, pp. 967-980.

[Smit78] Smith, A. J., "Sequential Program Prefetching in Memory Hierarchies." *IEEE Computer Magazine,* Vol. 11, No. 12, December 1978.

[Smit82] Smith, A. J., "Cache Memories." *ACM Computing Surveys,* Vol. 14, No. 3, September 1982, pp. 473-530.

[Smit85] Smith, A. J., "Cache Evaluation and the Impact of Workload Choice." To appear, Proceedings 12th Annual Symposium on Computer Architecture, Boston, MA, June 1985.